White Paper

Last Revision Date:
September 22, 2006

Author:
Rajshree Chabukswar
Applications Engineer
Software & Solutions Group

Maximizing Performance and Energy-Efficiency on Intel® Core™ Microarchitecture using Multi-Threading

Table of Contents

Table of Figures

1.0 Overview: Multithreading for Maximum Performance and Energy-Efficiency

With the launch of the Intel® Core™ Duo processor, Intel introduced the first dual-core processor in the mobile market segment. Since availability of execution resources on the system doubled, by virtue of two cores in that processor, many applications will adopt multithreading to take advantage of the new available CPU resources. Since that product launch, Intel has followed up with new multi-core processors based on the Intel® Core™ microarchitecture for the mobile, desktop, server, and workstation market segments. All of these market segments will require optimized threading models and techniques to obtain the highest level of performance and energy-efficiency that these multi-core processors offer.

1.1 *Purpose of This Document*

Performance and energy-efficiency has always been one of the major areas of importance for mobile platforms, and as such, a mobile platform makes a great test vehicle to uncover the relationship between (1) threading and performance and (2) threading and energy-efficiency. This paper will show that with multithreaded applications, the job at hand may be able to finish faster than single-threaded applications. As a result, the boost in performance may result in power savings as system resources will be used for less time, as compared to a single-threaded version. Fortunately, these concepts and corresponding benefits apply to not only mobile but the aforementioned desktop, server, and workstation market segments as well.

There are other considerations introduced with multithreading an application, such as the effects on power/performance when the threads in the application are imbalanced (such as when one thread does significantly more work than the other threads), differences in CPU utilization of the threads (for example, one thread might consume 100 percent CPU, while the other threads might consume 10-20 percent of the CPU), and when the threads are affinitized to a single core rather than running them on separate cores. This paper investigates such issues with a wide variety of multithreaded applications and multitasking scenarios, and proposes recommendations that should be considered when multithreading an application to obtain the highest level of performance and energy-efficiency that these multi-core processors offer.

1.2 *Test Platform*

All the tests described in this paper were conducted on dual-core Intel Core Duo engineering sample systems with the “Napa” platform. Power measurements were accomplished with Fluke NetDAQ*.

2.0 Testing Methodology and Threading Models

2.1 *Testing Methodology*

A variety of applications (single-threaded and multithreaded implementations), along with test kernels developed in-house, are characterized here for power/performance measurements. These applications include a variety of content creation applications, kernels from the gaming space, kernels using Intel® Integrated Performance Primitives (IPP), and office productivity applications.

Single-threaded and multithreaded implementations of the applications discussed here were tested with two power schemes that Microsoft Windows* provides. These allowed running the applications either at full-CPU frequency or with the power-saving feature enabled. Performing measurements with these power schemes help explain the impact on performance and power when power-saving is enabled, compared to running at full CPU frequency.

While running a system in power-saving ("Adaptive") mode, as explained below, Intel SpeedStep® technology reduces power consumption on the platform and enhances battery life on laptop computers. Intel SpeedStep technology allows a processor's core voltage and clock frequency to be reduced when CPU utilization is low, in order to save power without user intervention. Note that Intel SpeedStep technology is effective only when the system runs in the power saving (Adaptive) mode on Microsoft Windows* XP. Two power modes include:

1. MaxPerf: Also known as Always-On (AO) mode. This power scheme provides maximum performance. The processor with this power scheme runs at maximum available frequency, favoring performance over battery life.
2. Adaptive: Also known as Portable-Laptop (PL) mode. This power scheme is optimized to increase battery life. The processor frequency states are changed based on the workload of the processor, favoring battery life over performance. When the system is idle, the processor runs at the lowest processor frequency state in order to conserve power. The OS changes processor frequency states using Intel SpeedStep technology.

Before discussing the various experiments in detail, the next sections provide background information about the thread scheduling mechanism on Microsoft Windows as well as various threading models.

2.2 *Thread Scheduling*

On multiprocessor systems, the operating system (OS) usually schedules threads on different processors based on the system activity at any given time. A scheduling algorithm is typically used to schedule a thread on the processor when a processor is idle. When a new thread becomes ready for execution, it's either scheduled to execution directly or put in the ready queue depending on processor availability and priority of the new thread.

If an application doesn't specify "hard affinity" (referred to as affinitization here) for a particular processor/core, then the OS scheduler will schedule threads on any available processor using its scheduling algorithm. Microsoft offers application programming interfaces (APIs) to affinitize a thread to one particular processor, SetThreadAffinityMask(), which indicates that the thread will only be executed on an indicated processor—even when other processors are idle. Using affinitization may cause performance degradation, as the affinitized thread may be halted while waiting on the specified processor even when other processors on the system are idle.

To address this issue, Microsoft introduced another API, SetIdealProcessor(), which allows

developers to give a hint to the OS about which processor to prefer for execution of a particular thread. However, if the preferred processor is unavailable, the OS selects from among the other available processors using scheduling algorithms. The Results of Testing section below showcases studies including affinitize one or more threads to a single core to understand the effects on power/performance characteristics.

2.3 *Threading Models*

In essence there are two categories of threading models:

1. ***Data Domain Decomposition***: In this model, the available data set is divided into separate parts and each thread works on its individual portion. At the synchronization point, threads may combine their own work portions. Since each thread does the same work with a different data set, this threading model is less likely to have any performance impact if changes are done to the source code. The changes done in this case would impact both threads equally. For example, a data processing application that has two threads working on half of the available data set will execute similar instructions on its own data set. Any change done to the code base (new functionality added to filter some of the data) would impact equally on both threads since both the threads will have to run through the new functionality. As a result, multithreaded performance is less likely to be affected by the changes.
2. ***Functional Domain Decomposition:*** In this model, each thread works on separate functionality pieces/sections of code within an application. At synchronization points, threads usually combine their own work items. In this case, since each thread works on individual sections of code within an application, change in one or more sections may have impact on performance. For example, if the data processing application discussed above is multithreaded in such a way that one thread performs the data gathering phase and another thread performs the data manipulation phase, any changes made to data manipulation (new functionality added to filter some of the data) may cause imbalance among the two threads, as the data manipulation thread is likely to run for a longer time due to added functionality. Hence changes made in this case may have an impact on multithreaded performance.

Apart from data and functional decompositions, threading methodologies can be further divided into balanced and imbalanced threading as well.

1. ***Balanced threading model:*** In this case, each thread has an equal amount of work as other active threads of the application. An example of balanced threading is a data processing application (discussed in the Data Domain Decomposition section above) where two threads process half of the available data set by executing similar instructions on its own data set. It is the job of a developer to make sure that there is no dependency between the data being processed by separate threads. Data dependencies necessitate the use of synchronization objects, which significantly reduces the performance, producing minimal improvement in performance by use of multithreading techniques. It is always good to have minimal synchronization among threads. A balanced threading model usually produces better gains than an imbalanced model.
2. ***Imbalanced threading model:*** In this case, there is a significant difference in the amount of work done by each thread within an application. For example, in a data processing application one thread performance data gathering and the other thread performs data manipulation. If the data-gathering phase in the data processi ng application takes much less time than the data manipulation phase, it will result in thread imbalance. A functional threading model in general tends to create imbalanced threading in an application since each thread works on separate sections of the code, which may result in one thread finishing faster as compared to the other thread. This may reduce time spent running the parallel code, and may cause imbalance in the application. The Results section discusses the power/performance results of applications that are multithreading using one or more of the threading techniques discussed above.

3.0 Results of Testing

The graphs in this section discuss power/performance results on an Intel Core Duo engineering sample system running Windows* XP SP2. The numbers are in "seconds." Power measurements were done with Fluke NetDAQ, which reports average power (in watts (W)) which is then converted to total power using application run-time data (mWHr).

3.1 *Applications with balanced threading models*

As discussed in the Balanced Threading Model section above, the threads in such applications usually perform an almost equal amount of work and consume equal processing resources. Applications studied here are mostly CPU-intensive, consuming ~95-100 percent of the CPU. Here, we discuss the performance and power impact of such applications when run in single-threaded as well as multithreaded modes (referred as ST and MT, respectively). The performance data is measured in seconds, followed by CPU power consumption data and platform power consumption data measured in mWHr.

The graph in **Figure 1** indicates performance data for running ST and MT versions of the applications. Cryptography and Video Encoding applications have two MT implementations and results are indicated as MT-1 and MT-2. For content creation applications, multithreading is done with only one implementation, indicated as MT-1.

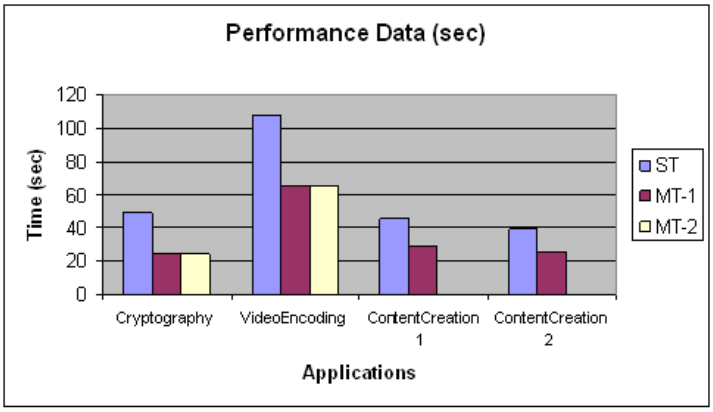

Figure 1. Balanced Threading Performance

As indicated in Figure 1, multithreaded applications clearly show significant performance improvements over running single-threaded versions. For example, the ST version of cryptography takes ~50 seconds to complete, while both the MT-1 and MT-2 versions take only ~25 seconds.

Now let's examine the effect of multithreading on power consumption. **Figures 2 and 3** indicate CPU and platform power for adaptive (portable/laptop) mode, respectively. Adaptive mode is chosen as it favors power consumption by dynamically changing CPU frequency on demand.

For each application run (ST and MT), power data-gathering is normalized to the longest run-time. For example, as indicated in Figure 1, the cryptography workload runs for ~50 seconds in ST mode and ~25 sec in MT modes. The power data is measured for 50 seconds in both ST and MT cases. The intent here is to determine if power can be saved by finishing the CPU-intensive task faster (as in the case of MT) and going to idle state for the remaining duration.

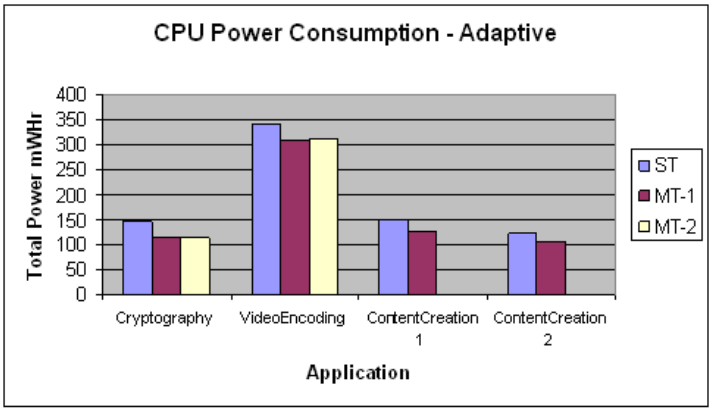

Figure 2. Balanced Threading - CPU Power (Adaptive)

As indicated in **Figure 2**, power saving is achieved by finishing the job faster (MT) and idling for the remainder of the time as the ST version. This indicates that multithreading done correctly not only shows performance improvements, but also saves power. For example, the cryptography ST version running for ~50 seconds consumes ~150 mWHr of total power, while running the cryptography MT version for ~25 seconds and idling the system for the remaining 25 seconds consumes ~110 mWHr of total power. Hence, multithreading helps save power.

The graph in **Figure 3** indicates the implication of multithreading on platform power.

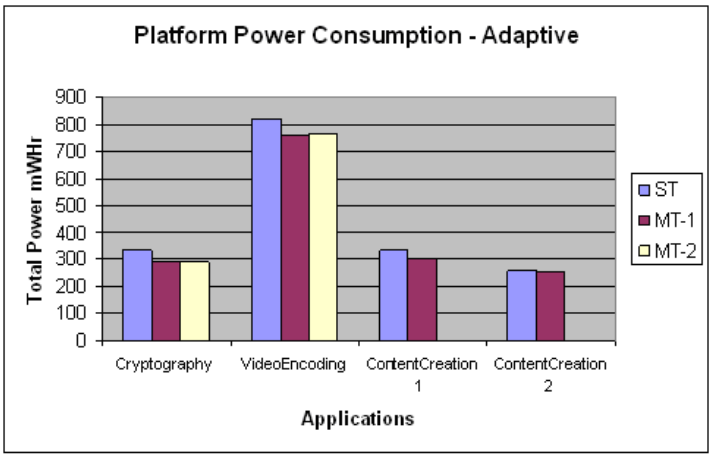

Figure 3. Balanced Threading - Platform Power (Adaptive)

As indicated, running a multithreaded version of an application consumes lower platform power compared to running a single-threaded version.

3.2 *Applications with an imbalanced threading model*

As demonstrated above, balanced multithreading provides performance as well as power-saving benefits, when compared to running a single-threaded version. In this section, we will examine power/performance implications on an application with an imbalanced threading model. For this study, a sample game physics engine was created (using Microsoft DirectX*). The sample application has two parts: 1) Physics Computation (collision detection and resolution for graphics objects), and 2) Rendering (updated positions are drawn onto screen). The design of the application was deliberate so that balanced and imbalanced threading could be studied for a CMP processor. Briefly:

- Balanced: For this implementation, graphical objects (and background imagery) were divided into two parts, and each thread takes care of the collision detection and resolution of its own set of objects.
- Imbalanced: In this implementation, one thread was tasked with performing collision detection and resolution for the colliding objects, while the other thread calculated the updated positions. The result was the desired goal of the first thread being more CPU-intensive than the second thread.

The intent behind creating two multithreaded implementations here is to evaluate power/performance impact when an imbalanced threading model is used, as compared to a balanced threading model.

With the two implementations, performance data in different power schemes (MaxPerf and Adaptive) are as shown below. Let us focus on the first two data sets (MaxPerf and Adaptive – Default) for now.

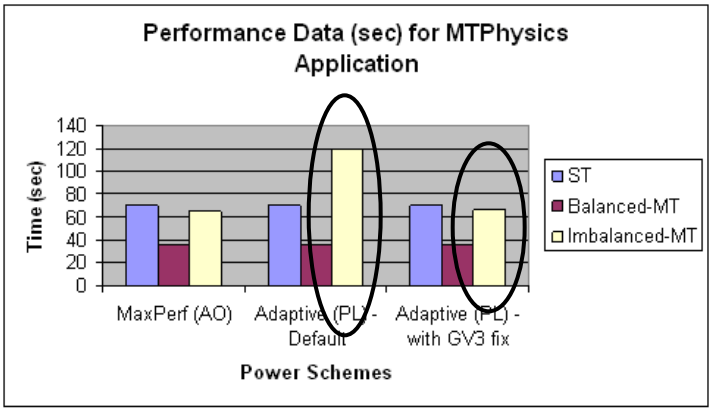

Figure 4. Imbalanced Threading Performance

As indicated on the first two sets in the graph above, performance of imbalanced multithreaded (Imbalanced-MT) implementation degrades from ~64 seconds in MaxPerf mode to ~120 seconds in Adaptive mode (indicated with circles).

Let's now look at what happens to platform power consumption with this decline in performance. The power measurements discussed here were normalized using a technique mentioned in the "Balanced Threading Model" section.

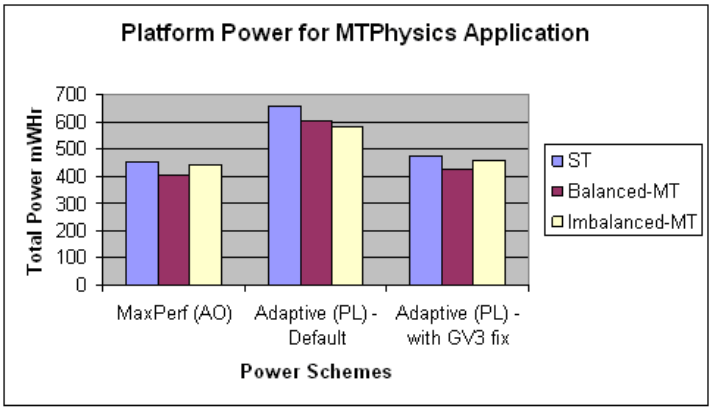

Figure 5. Imbalanced Threading - Platform Power

As indicated in the graph above (**Figure 5**), the platform power consumption increases with a reduction in the performance in Adaptive (PL) – Default mode. Since the Imbalanced-MT workload now takes much longer to finish, causing performance degradation, power consumption increases. (The second data set is normalized to run time of 'Imbalanced-MT' as that is the longest run-time for the second data set).

By looking at the application profile while running the imbalanced-MT version, it is observed that since only one of the threads is doing a large amount of the work, the first thread keeps migrating between the cores, making effective CPU utilization on the cores at ~50 percent. This is a natural result of the manner in which the Windows scheduler works. However, on a system running in Adaptive (portable/laptop) power mode, this thread migration causes the Windows kernel power manager to incorrectly calculate the optimal target performance state for the processor, as the individual cores may appear less busy than the whole package. Due to this, the Windows OS tends to reduce processor frequency in order to save power in adaptive mode, hence performance results may show degradation in adaptive mode. This, in turn, causes increased power consumption.

To address this issue of incorrectly calculating the optimal frequency, Microsoft provided a hotfix (KB896256) to change the kernel power manager to track CPU utilization across the entire package, rather than the individual cores, thus calculating the optimum frequency for applications. As a result, even when the thread keeps migrating, the CPU frequency reduction decision will be made by looking at the entire package (having both cores) and not individual cores.

The third set in Figure 4 indicates data with a kernel hotfix. In this case, imbalanced-MT implementation in Adaptive (PL) mode shows similar power/performance data as that of MaxPerf (AO) mode. With this fix, processors run at optimum frequency, not causing degradation in Adaptive (PL) mode. Platform power data with the fix is shown in Figure 5 in the Adaptive (PL) - with GV3 Fix column.

This study shows that the imbalanced threading model/under-utilized CPU may cause degradation in performance, causing increased power consumption. It is recommended to use a balanced threading model while multithreading applications. Thread imbalance can be identified by using tools like the Microsoft Perfmon*, the

timeline view offered by Intel® VTune™ and Intel® Threading Tools (such as Intel® Thread Checker and Intel® Thread Profiler) to track individual thread run-time and processor utilization counters.

3.3 *Multitasking scenarios with one application affinitized to single core*

One of the common usage scenarios for PC users is running multiple applications simultaneously — multitasking. To understand the performance and power impact of running two applications simultaneously, an experiment was conducted with two office productivity applications running concurrently. Since scheduling the two applications using different techniques is likely to show power/performance impacts, the following scenarios were examined:

1. Microsoft Windows XP scheduling both applications using its scheduling algorithm (no affinitization).
2. Each application hard-affinitized to each core. Application 1 runs on core 0 and application 2 runs on core 1.
3. One of the applications hard-affinitized to Core 0 while Windows XP schedules the other application.

These scheduling configurations were chosen to identify if a certain scheduling mechanism favors both power and performance as compared to the others.

Performance data for these configurations is shown in **Figure 6**. As indicated in the graph, there is no significant performance difference observed with different scheduling configurations.

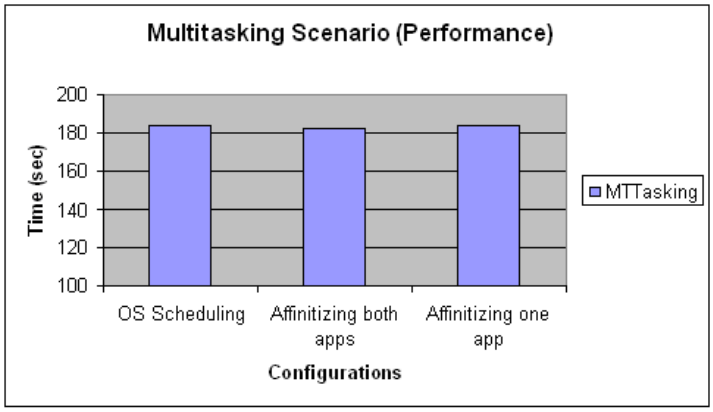

Figure 6. Multitasking Scenario Performance

As shown in **Figure 7**, CPU power consumption data—the second scenario that is affinitizing both applications to individual cores—demonstrates slightly higher power consumption as compared to Windows XP scheduling.

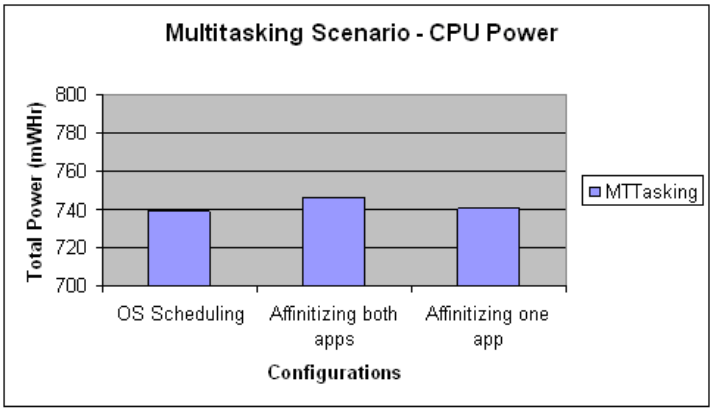

Figure 7. Multitasking Scenario - CPU Power

A similar impact is seen (**Figure 8**) on platform power consumption, where hard-affinitizing both the applications to each core shows slightly higher platform power consumption as compared to letting Windows XP do the scheduling.

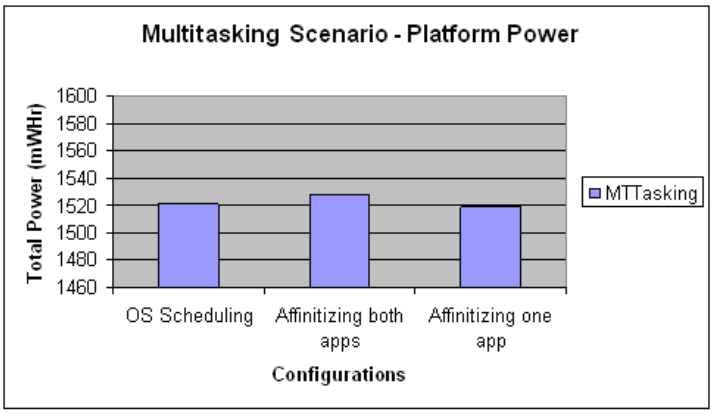

Figure 8. Multitasking Scenario - Platform Power

4.0 Summary

The following points can be deduced from the threading tests described in this article.

4.1 *Threading done right provides performance boosts as well as power savings*

As indicated by the data in the Balanced Multithreading Model section, a properly multithreaded application demonstrates performance improvement as well as power savings. This includes multithreaded implementations with a minimum of imbalance and synchronization points. While multithreading an application, it is always recommended to use a threading model in which all the threads perform an equal amount of work independently. Minimizing synchronization points between threads leads to more time spent in the parallel section, which translates to good performance improvements and power savings.

4.2 *Thread imbalance may cause performance degradation and may not provide power/performance benefits as compared to balanced threading*

As discussed in the Imbalanced Threading Model section, applications with an imbalanced threading model may show less performance improvement as compared to a balanced threading model, and thus consume more power as compared to the balanced implementation. Thread imbalances may cause frequent thread migration across cores, which may result in reporting incorrect processor activity states. This may lead processors to go down to lower frequency states (if the hotfix provided by Microsoft is not enabled) in adaptive schemes even if one of the threads is utilizing full-processor resources. This issue may occur while running single-threaded applications on a dual-core system in Adaptive mode as well.

4.3 *Utilize GV3 hotfix (KB896256) from Microsoft*

If a multithreaded application indicates performance degradation/increased power consumption in Adaptive (PL) mode, install GV3 hotfix from Microsoft; the issue might be that the OS is getting incorrect information about processor performance while in Adaptive mode. GV3 hotfix will track CPU utilization across the entire package rather than individual cores, and will enable the OS to operate at optimum frequency.

4.4 *OS scheduling vs. hard affinitizing*

In general, for Intel Core Duo systems, it is advisable to use OS scheduling as opposed to affinitizing threads or applications. The OS scheduler will utilize any under-utilized core, while hard affinitizing may potentially degrade performance since an application may need to wait for the availability of a specific processor even though other processors in the system are idle.

5.0 More Information and Author BIO

5.1 *More Info*

You can learn much more by visiting the following areas of the Intel® Software Network and the Energy-Efficient Performance websites:

- Intel® Core™2 Duo Processor
- Threading for Developers
- Intel Multi-Core Processing
- Energy-Efficient Performance

5.2 *Author Bio*

Rajshree Chabukswar is an applications engineer in Intel's Mobile Application Enabling Group – part of the Software & Solutions Group – working on client enabling. Prior to coming to Intel, she obtained an M.S. in Computer Engineering at Syracuse University.

[FOOTNOTE]

Performance/power tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components you are considering purchasing.